BOOK 12: WORKING WITH APIs
Chapter 1: What Are APIs

UNDERSTANDING APIs

API stands for Application Programming Interface.

An API is a way for two programs to talk to each other.

Think of a restaurant. You are the customer. The kitchen is the server. The waiter is the API. You tell the waiter what you want. The waiter tells the kitchen. The kitchen makes your food. The waiter brings it back to you.

You never go into the kitchen yourself. You use the waiter as an interface.

APIs work the same way. Your JavaScript code is the customer. A server somewhere has data or services. The API is how you communicate with that server.

WHY APIs MATTER

Without APIs, every application would be isolated.

With APIs you can:
- Get weather data from a weather service
- Show maps from a mapping service
- Process payments through a payment service
- Send emails through an email service
- Store files in cloud storage
- Authenticate users with social logins
- Access any public data source

APIs let you build on top of existing services instead of building

everything yourself.

TYPES OF APIs

Web APIs are the most common in JavaScript.

They use HTTP (the same protocol browsers use to load web pages) to send and receive data.

REST APIs are the most popular style. REST stands for Representational State Transfer. We will focus on REST APIs in this book.

Other types include:
- GraphQL (query-based, newer style)
- WebSocket (real-time, two-way communication)
- SOAP (older, XML-based, mostly enterprise)

For now, understand that when developers say API they usually mean REST API.

HOW WEB APIs WORK

Your code sends a request to a URL. The server processes the request. The server sends back a response.

The request includes:
- URL (where to send it)
- Method (what action to take)
- Headers (extra information)
- Body (data you are sending)

The response includes:
- Status code (did it work?)

- Headers (extra information)
- Body (the data you asked for)

This is called the request-response cycle.

HTTP METHODS

APIs use HTTP methods to indicate what action you want:

GET - Retrieve data
POST - Create new data
PUT - Replace existing data
PATCH - Update part of existing data
DELETE - Remove data

When you visit a website, your browser sends a GET request. When you submit a form, your browser might send a POST request.

API calls work the same way but return data instead of web pages.

STATUS CODES

The server responds with a status code telling you what happened:

200 - OK (success)
201 - Created (new resource created)
204 - No Content (success, nothing to return)

400 - Bad Request (you sent invalid data)
401 - Unauthorized (you need to log in)
403 - Forbidden (you do not have permission)
404 - Not Found (resource does not exist)

500 - Internal Server Error (server broke)

502 - Bad Gateway (server could not reach another server)
503 - Service Unavailable (server is overloaded)

Status codes starting with 2 mean success.
Status codes starting with 4 mean you made a mistake.
Status codes starting with 5 mean the server made a mistake.

DATA FORMATS

APIs send and receive data in a format both sides understand.

JSON is the most common format today. JSON stands for JavaScript Object Notation. It looks like JavaScript objects but is actually text.

JSON example:

```
{
 "name": "Alice",
 "age": 25,
 "active": true,
 "hobbies": ["reading", "coding"]
}
```

JSON uses double quotes for strings and property names.

Other formats include XML (older, more verbose) and plain text.

ENDPOINTS

An endpoint is a specific URL that does something specific.

Example API with multiple endpoints:

https://api.example.com/users (all users)

https://api.example.com/users/123 (user with ID 123)
https://api.example.com/users/123/posts (posts by user 123)
https://api.example.com/posts (all posts)
https://api.example.com/posts/456 (post with ID 456)

Each endpoint is a different resource.

The method determines the action:

GET /users - get all users
POST /users - create a user
GET /users/123 - get user 123
PUT /users/123 - update user 123
DELETE /users/123 - delete user 123

API DOCUMENTATION

Every API has documentation explaining:
- What endpoints exist
- What methods each endpoint accepts
- What data to send in requests
- What data comes back in responses
- How to authenticate
- Rate limits and restrictions

You cannot use an API without reading its documentation.

Good documentation includes examples for every endpoint.

PUBLIC VS PRIVATE APIs

Public APIs are available to anyone. Some are completely free. Some require registration. Some require payment.

Examples of public APIs:
- OpenWeatherMap (weather data)
- GitHub API (repositories, users)
- NASA API (space data)
- REST Countries (country information)

Private APIs are internal to a company. They connect the company's own systems.

You will work with both. The principles are the same.

YOUR FIRST API CALL

Let us make a real API call to a public API.

REST Countries provides country data for free.

```
fetch("https://restcountries.com/v3.1/name/japan")
 .then(response = > response.json())
 .then(data = > {
 console.log(data[0].name.common)
 console.log(data[0].capital[0])
 console.log(data[0].population)
 })
```

Output:
Japan
Tokyo
125836021

You just retrieved real data from a real server on the internet.

WHAT HAPPENS IN THAT CALL

fetch("https://restcountries.com/v3.1/name/japan")

This sends a GET request to that URL.

.then(response => response.json())

The response comes back. We convert the body from JSON text to a JavaScript object.

.then(data => {...})

Now we have the actual data and can use it.

The API returned an array. The first element has the country data. We access name.common, capital[0], and population.

PRACTICE

Understand APIs are how programs communicate.

Understand the request-response cycle.

Know the common HTTP methods and status codes.

Understand JSON as a data format.

Know what endpoints are.

Try the REST Countries API call in your browser console.

WHAT YOU LEARNED

You learned APIs let programs communicate.
You learned the request-response cycle.

You learned HTTP methods indicate actions.
You learned status codes indicate results.
You learned JSON is the common data format.
You learned endpoints are specific URLs for specific resources.

The next chapter teaches you HTTP in more detail.

Chapter 2: HTTP Basics

WHAT IS HTTP

HTTP stands for HyperText Transfer Protocol.

It is the foundation of data communication on the web.

Every time you load a webpage, your browser uses HTTP. Every time you make an API call, you use HTTP.

HTTPS is HTTP with encryption. The S stands for Secure. Most modern APIs require HTTPS.

THE REQUEST

An HTTP request has these parts:

Method - What action to take (GET, POST, etc.)
URL - Where to send the request
Headers - Extra information about the request
Body - Data you are sending (for POST, PUT, PATCH)

Example request:

```
POST /users HTTP/1.1
```

Host: api.example.com
Content-Type: application/json
Authorization: Bearer abc123

{"name": "Alice", "email": "alice@example.com"}

The first line has the method and path.
The next lines are headers.
The blank line separates headers from body.
The last part is the body.

THE RESPONSE

An HTTP response has these parts:

Status Line - Protocol version and status code
Headers - Extra information about the response
Body - The data being returned

Example response:

HTTP/1.1 201 Created
Content-Type: application/json
Location: /users/123

{"id": 123, "name": "Alice", "email": "alice@example.com"}

The first line has the status code and message.
The next lines are headers.
After the blank line is the body.

HTTP METHODS IN DETAIL

GET retrieves data.

GET /users/123

No body. The server returns the requested resource.

POST creates new data.

POST /users
Body: {"name": "Alice"}

The body contains the data to create. The server returns the created resource.

PUT replaces existing data completely.

PUT /users/123
Body: {"name": "Alice", "email": "alice@example.com", "age": 25}

You must send all fields. Missing fields may be deleted.

PATCH updates part of existing data.

PATCH /users/123
Body: {"age": 26}

Only the fields you send are updated. Other fields stay the same.

DELETE removes data.

DELETE /users/123

Usually no body needed. The server removes the resource.

STATUS CODES IN DETAIL

Success codes (2xx):

200 OK - Request succeeded
201 Created - New resource created
204 No Content - Success but nothing to return
206 Partial Content - Part of a resource returned

Client error codes (4xx):

400 Bad Request - Invalid syntax or data
401 Unauthorized - Authentication required
403 Forbidden - You do not have permission
404 Not Found - Resource does not exist
405 Method Not Allowed - Wrong method for this endpoint
409 Conflict - Request conflicts with current state
422 Unprocessable Entity - Valid syntax but invalid data
429 Too Many Requests - Rate limit exceeded

Server error codes (5xx):

500 Internal Server Error - Generic server error
502 Bad Gateway - Server received invalid response from upstream
503 Service Unavailable - Server temporarily unavailable
504 Gateway Timeout - Upstream server did not respond

COMMON HEADERS

Request headers you send:

Content-Type: application/json
Tells the server what format your body is in.

Accept: application/json
Tells the server what format you want back.

Authorization: Bearer token123
Provides authentication credentials.

User-Agent: MyApp/1.0
Identifies your application.

Response headers you receive:

Content-Type: application/json
What format the body is in.

Content-Length: 1234
Size of the body in bytes.

Cache-Control: max-age=3600
How long to cache the response.

X-RateLimit-Remaining: 95
How many requests you have left.

QUERY PARAMETERS

Add data to GET requests via the URL:

GET /users?page=2&limit=10&sort=name

The question mark starts the query string.
Parameters are key=value pairs.
Ampersand separates parameters.

Use query parameters for:
- Filtering (status=active)
- Sorting (sort=date)
- Pagination (page=2&limit=10)
- Searching (q=searchterm)

PATH PARAMETERS

Include data in the URL path:

GET /users/123
GET /users/123/posts/456

The values 123 and 456 are path parameters.

Use path parameters for identifying specific resources.

Path parameters identify WHAT you want.
Query parameters modify HOW you get it.

REQUEST BODY FORMATS

JSON is most common:

Content-Type: application/json

{"name": "Alice", "age": 25}

Form data for traditional forms:

Content-Type: application/x-www-form-urlencoded

name=Alice&age=25

Multipart for file uploads:

Content-Type: multipart/form-data

--boundary
Content-Disposition: form-data; name="file"; filename="photo.jpg"
Content-Type: image/jpeg

[binary file data]
--boundary--

IDEMPOTENCY

An operation is idempotent if doing it multiple times has the same effect as doing it once.

GET is idempotent. Getting a resource twice returns the same result.

PUT is idempotent. Replacing a resource twice leaves it in the same state.

DELETE is idempotent. Deleting something twice means it is still deleted.

POST is NOT idempotent. Creating a resource twice creates two resources.

PATCH may or may not be idempotent depending on the operation.

This matters for retrying failed requests. Safe to retry idempotent operations.

STATELESSNESS

HTTP is stateless. Each request is independent.

The server does not remember previous requests.

If you need the server to know who you are, you must include that information in every request. Usually via an Authorization header.

This is why you send tokens with every API call.

CORS

CORS stands for Cross-Origin Resource Sharing.

Browsers block requests to different domains for security.

If your page is at example.com, you cannot normally fetch from api.other.com.

The API server must send CORS headers allowing your domain.

Access-Control-Allow-Origin: *

This header says any domain can access the API.

CORS only affects browsers. Server-side code does not have this restriction.

PRACTICAL EXAMPLE

Building a complete request:

```
let url = "https://api.example.com/users"
let method = "POST"
let headers = {
 "Content-Type": "application/json",
 "Authorization": "Bearer mytoken123"
}
let body = JSON.stringify({
```

```
 name: "Alice",
 email: "alice@example.com"
})

fetch(url, {
 method: method,
 headers: headers,
 body: body
})
.then(response => {
 console.log("Status:", response.status)
 console.log("OK:", response.ok)
 return response.json()
})
.then(data => {
 console.log("Response:", data)
})
```

This sends a POST request with authentication to create a new user.

PRACTICE

Understand the structure of requests and responses.

Know all HTTP methods and when to use each.

Memorize common status codes.

Understand headers for content type and authorization.

Know the difference between query and path parameters.

WHAT YOU LEARNED

You learned HTTP request structure.
You learned HTTP response structure.
You learned methods in detail.
You learned status codes in detail.
You learned common headers.
You learned query and path parameters.
You learned about idempotency and statelessness.

The next chapter covers the Fetch API in depth.

BOOK 12: WORKING WITH APIs

Chapter 3: Fetch API Deep Dive

THE FETCH FUNCTION

fetch is the modern way to make HTTP requests in JavaScript.

```
fetch(url)
fetch(url, options)
```

It returns a Promise that resolves to a Response object.

```
let response = await fetch("https://api.example.com/data")
```

BASIC GET REQUEST

```
let response = await fetch("https://api.example.com/users")
let data = await response.json()

console.log(data)
```

The first await gets the response.
The second await parses the body as JSON.

THE RESPONSE OBJECT

The Response object has these properties:

response.status - HTTP status code (200, 404, etc.)
response.statusText - Status message ("OK", "Not Found")
response.ok - True if status is 200-299
response.headers - Response headers
response.url - Final URL after redirects
response.type - Response type
response.redirected - True if request was redirected

```
let response = await fetch(url)

console.log(response.status)
console.log(response.ok)
console.log(response.headers.get("Content-Type"))
```

READING THE BODY

The body can be read in different formats:

response.json() - Parse as JSON (returns Promise)
response.text() - Read as plain text (returns Promise)
response.blob() - Read as binary blob (returns Promise)
response.arrayBuffer() - Read as raw bytes (returns Promise)
response.formData() - Read as FormData (returns Promise)

Each method returns a Promise.

The body can only be read once. Reading it again throws an error.

```
let data = await response.json()
```

POST REQUEST

```
let response = await fetch("https://api.example.com/users", {
 method: "POST",
 headers: {
 "Content-Type": "application/json"
 },
 body: JSON.stringify({
 name: "Alice",
 email: "alice@example.com"
 })
})

let newUser = await response.json()

console.log(newUser)
```

The options object contains method, headers, and body.

PUT REQUEST

```
let response = await fetch("https://api.example.com/users/123", {
 method: "PUT",
 headers: {
 "Content-Type": "application/json"
 },
 body: JSON.stringify({
 name: "Alice Updated",
 email: "alice@example.com",
 age: 26
 })
})
```

PATCH REQUEST

```
let response = await fetch("https://api.example.com/users/123", {
 method: "PATCH",
 headers: {
 "Content-Type": "application/json"
 },
 body: JSON.stringify({
 age: 27
 })
})
```

DELETE REQUEST

```
let response = await fetch("https://api.example.com/users/123", {
 method: "DELETE"
})

if (response.ok) {
 console.log("User deleted")
}
```

REQUEST HEADERS

```
let response = await fetch(url, {
 headers: {
 "Content-Type": "application/json",
 "Accept": "application/json",
 "Authorization": "Bearer token123",
 "X-Custom-Header": "custom value"
 }
})
```

Headers can be an object or a Headers instance.

```
let headers = new Headers()
headers.append("Content-Type", "application/json")
headers.append("Authorization", "Bearer token123")

let response = await fetch(url, { headers })
```

READING RESPONSE HEADERS

```
let contentType = response.headers.get("Content-Type")
let remaining = response.headers.get("X-RateLimit-Remaining")

response.headers.forEach((value, name) => {
 console.log(name, value)
})
```

QUERY PARAMETERS

Build URLs with query parameters:

```
let params = new URLSearchParams({
 page: 2,
 limit: 10,
 sort: "name"
})

let url = `https://api.example.com/users?${params}`

let response = await fetch(url)
```

URLSearchParams handles encoding special characters.

```
let params = new URLSearchParams()
params.append("search", "hello world")
params.append("filter", "status=active")
```

ERROR HANDLING

fetch only rejects on network errors, not HTTP errors.

A 404 response is still a successful fetch.

```
async function fetchData(url) {
 let response = await fetch(url)

 if (!response.ok) {
 throw new Error(`HTTP error: ${response.status}`)
 }

 return response.json()
}
```

Always check response.ok before parsing the body.

TIMEOUT WITH ABORT CONTROLLER

fetch has no built-in timeout. Use AbortController.

```
let controller = new AbortController()

let timeoutId = setTimeout(() => controller.abort(), 5000)

try {
 let response = await fetch(url, {
 signal: controller.signal
 })
```

```
 clearTimeout(timeoutId)

 let data = await response.json()
} catch (error) {
 if (error.name === "AbortError") {
 console.log("Request timed out")
 } else {
 throw error
 }
}
```

CANCELING REQUESTS

Use AbortController to cancel requests:

```
let controller = new AbortController()

fetch(url, { signal: controller.signal })
 .then(response => response.json())
 .then(data => console.log(data))
 .catch(error => {
 if (error.name === "AbortError") {
 console.log("Request was canceled")
 }
 })

controller.abort()
```

REQUEST MODE

Control CORS behavior:

```
fetch(url, { mode: "cors" })
```

fetch(url, { mode: "no-cors" })
fetch(url, { mode: "same-origin" })

cors - Default. Requires CORS headers from server.
no-cors - Limited request. Response is opaque.
same-origin - Only same-origin requests allowed.

CREDENTIALS

Control cookie handling:

fetch(url, { credentials: "omit" })
fetch(url, { credentials: "same-origin" })
fetch(url, { credentials: "include" })

omit - Never send cookies.
same-origin - Default. Send cookies only to same origin.
include - Always send cookies, even cross-origin.

CACHE CONTROL

Control caching behavior:

fetch(url, { cache: "default" })
fetch(url, { cache: "no-store" })
fetch(url, { cache: "reload" })
fetch(url, { cache: "no-cache" })
fetch(url, { cache: "force-cache" })

default - Normal cache behavior.
no-store - Do not cache.
reload - Ignore cache, always fetch.
no-cache - Validate with server before using cache.
force-cache - Use cache even if stale.

REDIRECT HANDLING

Control redirect behavior:

```
fetch(url, { redirect: "follow" })
fetch(url, { redirect: "error" })
fetch(url, { redirect: "manual" })
```

follow - Default. Automatically follow redirects.
error - Throw error on redirect.
manual - Do not follow. Return redirect response.

COMPLETE FETCH WRAPPER

A reusable function for API calls:

```
async function api(endpoint, options = {}) {
 let url = `https://api.example.com${endpoint}`

 let config = {
 method: options.method ?? "GET",
 headers: {
 "Content-Type": "application/json",
 ...options.headers
 }
 }

 if (options.body) {
 config.body = JSON.stringify(options.body)
 }

 let response = await fetch(url, config)
```

```
 if (!response.ok) {
 let error = await response.json().catch(() => ({}))
 throw new Error(error.message ?? `HTTP ${response.status}`)
 }

 if (response.status === 204) {
 return null
 }

 return response.json()
}

let users = await api("/users")
let user = await api("/users", { method: "POST", body: { name:
"Alice" } })
await api("/users/123", { method: "DELETE" })
```

PRACTICE

Use fetch for GET, POST, PUT, PATCH, DELETE.

Always check response.ok before parsing.

Use URLSearchParams for query strings.

Use AbortController for timeouts.

Create reusable fetch wrappers.

WHAT YOU LEARNED

You learned the fetch function and its options.
You learned to read response status and body.
You learned all HTTP methods with fetch.

You learned to handle headers.
You learned error handling with fetch.
You learned about AbortController for timeouts.
You learned cache and credential options.

The next chapter teaches you to work with JSON.

BOOK 12: WORKING WITH APIs

Chapter 4: Working with JSON

WHAT IS JSON

JSON stands for JavaScript Object Notation.

It is a text format for storing and exchanging data.

JSON looks like JavaScript objects and arrays, but it is pure text.

```
{
 "name": "Alice",
 "age": 25,
 "active": true,
 "address": {
 "city": "New York",
 "country": "USA"
 },
 "hobbies": ["reading", "coding"]
}
```

JSON is the standard format for API communication.

JSON SYNTAX RULES

Property names must be in double quotes:

{"name": "Alice"}

Not: {name: "Alice"}
Not: {'name': 'Alice'}

String values must be in double quotes:

{"name": "Alice"}

Not: {"name": 'Alice'}

No trailing commas:

{"a": 1, "b": 2}

Not: {"a": 1, "b": 2,}

No comments:

{"name": "Alice"}

Not: {"name": "Alice" // this is Alice}

No undefined:

{"value": null}

Not: {"value": undefined}

JSON DATA TYPES

JSON supports these types:

String - "hello"
Number - 123, 3.14, -5, 1e10
Boolean - true, false
Null - null
Object - {"key": "value"}
Array - [1, 2, 3]

JSON does NOT support:

undefined
Functions
Dates (must be strings)
Infinity and NaN
Symbol
BigInt

JSON.stringify

Convert JavaScript to JSON string:

```
let user = {
 name: "Alice",
 age: 25,
 active: true
}

let json = JSON.stringify(user)

console.log(json)
```

Output: {"name":"Alice","age":25,"active":true}

The result is a string, not an object.

```
console.log(typeof json)
```

Output: string

STRINGIFY OPTIONS

Pretty print with indentation:

```
let json = JSON.stringify(user, null, 2)

console.log(json)
```

Output:

```
{
 "name": "Alice",
 "age": 25,
 "active": true
}
```

The third argument is the number of spaces to indent.

Filter properties with a replacer:

```
let json = JSON.stringify(user, ["name", "age"])
```

Output: {"name":"Alice","age":25}

Only specified properties are included.

Transform values with a replacer function:

```
let json = JSON.stringify(user, (key, value) => {
 if (typeof value === "string") {
 return value.toUpperCase()
 }
 return value
})
```

Output: {"name":"ALICE","age":25,"active":true}

JSON.parse

Convert JSON string to JavaScript:

```
let json = '{"name":"Alice","age":25}'

let user = JSON.parse(json)

console.log(user.name)
console.log(user.age)
```

Output:
Alice
25

Now user is a JavaScript object you can work with.

PARSE WITH REVIVER

Transform values while parsing:

```
let json = '{"name":"Alice","birthDate":"1998-05-15"}'

let user = JSON.parse(json, (key, value) = > {
 if (key = = = "birthDate") {
 return new Date(value)
 }
 return value
})

console.log(user.birthDate instanceof Date)
```

Output: true

The reviver function receives each key-value pair.

HANDLING PARSE ERRORS

JSON.parse throws on invalid JSON:

```
try {
 let data = JSON.parse("not valid json")
} catch (error) {
 console.log("Invalid JSON:", error.message)
}
```

Output: Invalid JSON: Unexpected token 'o'

Always wrap JSON.parse in try-catch when parsing unknown input.

SAFE PARSE FUNCTION

```
function safeJsonParse(text, fallback = null) {
 try {
 return JSON.parse(text)
 } catch {
 return fallback
 }
}

let data = safeJsonParse('{"valid": true}')
let invalid = safeJsonParse("not json", {})

console.log(data)
console.log(invalid)
```

Output:
{valid: true}
{}

DATES IN JSON

JSON has no date type. Dates become strings:

```
let event = {
 name: "Meeting",
 date: new Date("2024-03-15")
}

let json = JSON.stringify(event)

console.log(json)
```

Output: {"name":"Meeting","date":"2024-03-15T00:00:00.000Z"}

The date is converted to ISO string format.

Parse dates back:

```
let parsed = JSON.parse(json)

console.log(typeof parsed.date)
```

Output: string

The date is still a string. Convert it manually:

```
parsed.date = new Date(parsed.date)
```

SPECIAL VALUES IN JSON

Some values cannot be represented:

```
let obj = {
 func: function() {},
 undef: undefined,
 inf: Infinity,
 nan: NaN,
 sym: Symbol("test")
}

let json = JSON.stringify(obj)

console.log(json)
```

Output: {"inf":null,"nan":null}

Functions, undefined, and symbols are omitted.
Infinity and NaN become null.

CIRCULAR REFERENCES

JSON.stringify fails on circular references:

```
let obj = { name: "Alice" }
obj.self = obj

JSON.stringify(obj)
```

Error: Converting circular structure to JSON

Handle with a replacer:

```
function getCircularReplacer() {
 let seen = new WeakSet()
```

```
 return (key, value) = > {
 if (typeof value = = = "object" && value ! = = null) {
 if (seen.has(value)) {
 return undefined
 }
 seen.add(value)
 }
 return value
 }
}

let json = JSON.stringify(obj, getCircularReplacer())
```

DEEP CLONE WITH JSON

Clone objects using JSON:

```
let original = {
 name: "Alice",
 address: {
 city: "NYC"
 }
}

let clone = JSON.parse(JSON.stringify(original))

clone.address.city = "LA"

console.log(original.address.city)
```

Output: NYC

The clone is completely independent.

Limitations:
- Loses functions
- Loses undefined

- Converts dates to strings
- Fails on circular references
- Slow for large objects

For better cloning, use structuredClone in modern browsers.

WORKING WITH API RESPONSES

Typical pattern with fetch:

```
async function getUser(id) {
 let response = await fetch(`https://api.example.com/users/${id}`)

 if (!response.ok) {
 throw new Error(`HTTP error: ${response.status}`)
 }

 let user = await response.json()

 return user
}
```

response.json() parses the response body as JSON.

SENDING JSON TO APIs

```
async function createUser(userData) {
 let response = await fetch("https://api.example.com/users", {
 method: "POST",
 headers: {
 "Content-Type": "application/json"
 },
 body: JSON.stringify(userData)
 })
```

```
 if (!response.ok) {
 let error = await response.json()
 throw new Error(error.message)
 }

 return response.json()
}

let newUser = await createUser({
 name: "Alice",
 email: "alice@example.com"
})
```

VALIDATING JSON STRUCTURE

Check that API response has expected shape:

```
function validateUser(data) {
 if (typeof data !== "object" || data === null) {
 throw new Error("Invalid user: not an object")
 }

 if (typeof data.name !== "string") {
 throw new Error("Invalid user: name must be string")
 }

 if (typeof data.email !== "string") {
 throw new Error("Invalid user: email must be string")
 }

 return data
}

let response = await fetch(url)
let data = await response.json()
let user = validateUser(data)
```

Never trust API data. Validate before using.

PRACTICE

Use JSON.stringify to convert objects to JSON.

Use JSON.parse to convert JSON to objects.

Handle parse errors with try-catch.

Understand date handling in JSON.

Validate API response structure.

WHAT YOU LEARNED

You learned JSON syntax rules.
You learned JSON.stringify and its options.
You learned JSON.parse and revivers.
You learned to handle parse errors.
You learned about dates and special values.
You learned about circular reference handling.
You learned to validate JSON structure.

The next chapter teaches you about API authentication.
BOOK 12: WORKING WITH APIs
Chapter 5: API Authentication

WHY AUTHENTICATION

APIs need to know who is making requests.

Without authentication:
- Anyone can access private data
- No way to limit usage per user
- No way to track who did what
- No personalization

Authentication proves you are who you claim to be.
Authorization determines what you are allowed to do.

API KEYS

The simplest form of authentication.

An API key is a long string you include with requests.

You get it from the API provider after registration.

In query parameter:

```
fetch("https://api.example.com/data?api_key=abc123xyz")
```

In header:

```
fetch("https://api.example.com/data", {
 headers: {
 "X-API-Key": "abc123xyz"
 }
})
```

API keys identify your application but not individual users.

API KEY SECURITY

Never expose API keys in client-side code.

Anyone can view your JavaScript source.

If your key is visible:
- Others can use your quota
- You could be charged for their usage
- Your key might be revoked

For browser apps, route requests through your own server. Your server has the API key. Browsers talk to your server.

Some APIs provide publishable keys for client-side use. These have limited permissions.

BASIC AUTHENTICATION

Username and password in every request.

Credentials are Base64 encoded (not encrypted).

```
let credentials = btoa("username:password")

fetch("https://api.example.com/data", {
 headers: {
 "Authorization": `Basic ${credentials}`
 }
})
```

btoa converts string to Base64.

Basic auth is simple but sends credentials with every request. Only use over HTTPS.

BEARER TOKENS

The most common modern approach.

First, authenticate to get a token.
Then, send the token with every request.

```
async function login(email, password) {
 let response = await fetch("https://api.example.com/login", {
 method: "POST",
 headers: {
 "Content-Type": "application/json"
 },
 body: JSON.stringify({ email, password })
 })

 let data = await response.json()

 return data.token
}

let token = await login("alice@example.com", "password123")
```

Use the token:

```
fetch("https://api.example.com/users/me", {
 headers: {
 "Authorization": `Bearer ${token}`
 }
})
```

Bearer means "the bearer of this token is authorized."

JWT TOKENS

JWT stands for JSON Web Token.

A JWT has three parts separated by dots:

header.payload.signature

Example:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEyMywiZXhwIjo

The header specifies the algorithm.
The payload contains user data and expiration.
The signature proves it was not tampered with.

You can decode the payload (it is just Base64):

```
let payload = token.split(".")[1]
let decoded = JSON.parse(atob(payload))

console.log(decoded)
```

Output: {userId: 123, exp: 1704067200}

exp is expiration time in seconds since 1970.

TOKEN STORAGE

Where to store tokens in browsers:

Memory (variable):
- Safest from XSS
- Lost on page refresh
- Lost on new tab

localStorage:
- Persists across sessions
- Vulnerable to XSS attacks

- Shared across tabs

sessionStorage:
- Cleared when browser closes
- Vulnerable to XSS attacks
- Not shared across tabs

HttpOnly Cookie:
- Not accessible from JavaScript
- Safest from XSS
- Requires server coordination
- Vulnerable to CSRF

For most apps, localStorage is acceptable with good XSS protection.

TOKEN REFRESH

Tokens expire for security.

Short-lived access tokens (minutes to hours).
Long-lived refresh tokens (days to weeks).

When access token expires, use refresh token to get a new one.

```
async function refreshAccessToken() {
 let refreshToken = localStorage.getItem("refreshToken")

 let response = await fetch("https://api.example.com/refresh", {
 method: "POST",
 headers: {
 "Content-Type": "application/json"
 },
 body: JSON.stringify({ refreshToken })
 })

 if (!response.ok) {
 localStorage.removeItem("refreshToken")
```

```
 throw new Error("Session expired")
 }

 let data = await response.json()

 localStorage.setItem("accessToken", data.accessToken)

 return data.accessToken
}
```

OAUTH 2.0

OAuth lets users log in with Google, GitHub, Facebook, etc.

Flow:

1. User clicks "Login with Google"
2. Redirect to Google login page
3. User logs in with Google
4. Google redirects back with authorization code
5. Your server exchanges code for tokens
6. Your server creates a session

The redirect URL includes the code:

https://yourapp.com/callback?code=abc123

Your server sends this code to Google (with your client secret) to get the actual tokens.

OAuth is complex. Use a library or auth service.

IMPLEMENTING LOGIN

```
class AuthService {
 constructor(baseUrl) {
 this.baseUrl = baseUrl
 this.token = localStorage.getItem("token")
 }

 async login(email, password) {
 let response = await fetch(`${this.baseUrl}/login`, {
 method: "POST",
 headers: { "Content-Type": "application/json" },
 body: JSON.stringify({ email, password })
 })

 if (!response.ok) {
 let error = await response.json()
 throw new Error(error.message ?? "Login failed")
 }

 let data = await response.json()

 this.token = data.token
 localStorage.setItem("token", data.token)

 return data.user
 }

 logout() {
 this.token = null
 localStorage.removeItem("token")
 }

 isLoggedIn() {
 return this.token !== null
 }

 getAuthHeaders() {
 if (!this.token) {
 return {}
 }
 return { "Authorization": `Bearer ${this.token}` }
```

```
 }
}

let auth = new AuthService("https://api.example.com")
```

AUTHENTICATED REQUESTS

Use the auth service for all API calls:

```
async function fetchWithAuth(url, options = {}) {
 let headers = {
 ...options.headers,
 ...auth.getAuthHeaders()
 }

 let response = await fetch(url, { ...options, headers })

 if (response.status === 401) {
 auth.logout()
 window.location.href = "/login"
 return
 }

 return response
}

let response = await fetchWithAuth("https://api.example.com/
users/me")
```

HANDLING UNAUTHORIZED

What to do when authentication fails:

```
async function apiRequest(url, options = {}) {
```

```
let response = await fetchWithAuth(url, options)

if (response.status === 401) {
let refreshed = await tryRefreshToken()

if (refreshed) {
return apiRequest(url, options)
}

redirectToLogin()
return
}

if (response.status === 403) {
throw new Error("You do not have permission")
}

return response
}
```

401 means not authenticated. Try to refresh or redirect to login.
403 means authenticated but not authorized.

SECURE PRACTICES

Never log tokens:

```
console.log(token)
```

Never include tokens in URLs:

```
fetch(`/api/data?token=${token}`)
```

Use HTTPS only:

```
fetch("https://api.example.com/...")
```

Set token expiration:
Short-lived tokens limit damage if stolen.

Validate tokens on every request:
Server must verify signature and expiration.

Revoke tokens on logout:
Tell the server the token is no longer valid.

PRACTICE

Understand API keys and when to use them.

Implement Bearer token authentication.

Store tokens securely.

Handle token expiration and refresh.

Handle unauthorized responses.

WHAT YOU LEARNED

You learned why APIs need authentication.
You learned API keys and their limitations.
You learned Bearer token authentication.
You learned about JWT token structure.
You learned token storage options.
You learned token refresh patterns.
You learned OAuth basics.

The next chapter teaches you API error handling.
BOOK 12: WORKING WITH APIs
Chapter 6: API Error Handling

TYPES OF ERRORS

API errors come from different sources:

Network errors - No internet, server down, DNS failure
HTTP errors - 4xx and 5xx status codes
Parse errors - Invalid JSON in response
Application errors - Business logic failures
Timeout errors - Request took too long

Each type needs different handling.

NETWORK ERRORS

fetch rejects on network failure:

```
try {
 let response = await fetch("https://api.example.com/data")
} catch (error) {
 if (error instanceof TypeError) {
 console.log("Network error:", error.message)
 }
}
```

Network errors throw TypeError with messages like:
- "Failed to fetch"
- "NetworkError when attempting to fetch resource"
- "Network request failed"

Check if user is online:

```
if (!navigator.onLine) {
```

```
 console.log("You are offline")
}
```

HTTP ERRORS

fetch does NOT throw on HTTP errors. A 404 is still a successful fetch.

```
let response = await fetch(url)

if (!response.ok) {
 console.log("HTTP error:", response.status)
}
```

response.ok is true for status 200-299.

You must check manually and throw if needed:

```
async function fetchData(url) {
 let response = await fetch(url)

 if (!response.ok) {
 throw new Error(`HTTP error: ${response.status}`)
 }

 return response.json()
}
```

ERROR RESPONSE BODIES

APIs often include error details in the response body:

```
{
 "error": {
```

```
 "code": "VALIDATION_ERROR",
 "message": "Email is required",
 "field": "email"
 }
}
```

Extract and use this information:

```
async function fetchData(url) {
 let response = await fetch(url)

 if (!response.ok) {
 let errorData = await response.json().catch(() => ({}))

 let error = new Error(errorData.message ?? `HTTP
${response.status}`)
 error.status = response.status
 error.code = errorData.code
 error.data = errorData

 throw error
 }

 return response.json()
}
```

CUSTOM API ERROR CLASS

```
class ApiError extends Error {
 constructor(message, status, data = {}) {
 super(message)
 this.name = "ApiError"
 this.status = status
 this.code = data.code
 this.data = data
 }
```

```
 get isClientError() {
 return this.status >= 400 && this.status < 500
 }

 get isServerError() {
 return this.status >= 500
 }

 get isAuthError() {
 return this.status === 401 || this.status === 403
 }

 get isNotFound() {
 return this.status === 404
 }

 get isValidationError() {
 return this.status === 400 || this.status === 422
 }
}
```

Use it:

```
async function fetchData(url) {
 let response = await fetch(url)

 if (!response.ok) {
 let data = await response.json().catch(() => ({}))
 throw new ApiError(
 data.message ?? response.statusText,
 response.status,
 data
 )
 }

 return response.json()
}
```

HANDLING SPECIFIC ERRORS

```
try {
 let data = await fetchData("/api/users/123")
} catch (error) {
 if (error instanceof ApiError) {
 if (error.isNotFound) {
 showMessage("User not found")
 } else if (error.isAuthError) {
 redirectToLogin()
 } else if (error.isValidationError) {
 showValidationErrors(error.data.errors)
 } else if (error.isServerError) {
 showMessage("Server error. Please try again later.")
 } else {
 showMessage("Something went wrong")
 }
 } else {
 showMessage("Network error. Check your connection.")
 }
}
```

TIMEOUT HANDLING

```
async function fetchWithTimeout(url, options = {}, timeout =
10000) {
 let controller = new AbortController()
 let timeoutId = setTimeout(() => controller.abort(), timeout)

 try {
 let response = await fetch(url, {
 ...options,
 signal: controller.signal
 })

 clearTimeout(timeoutId)
```

```
 return response
 } catch (error) {
 clearTimeout(timeoutId)

 if (error.name === "AbortError") {
 throw new Error("Request timed out")
 }

 throw error
 }
}
```

RETRY LOGIC

```
async function fetchWithRetry(url, options = {}, maxRetries = 3) {
 let lastError

 for (let attempt = 1; attempt <= maxRetries; attempt++) {
 try {
 let response = await fetch(url, options)

 if (response.ok) {
 return response
 }

 if (response.status >= 400 && response.status < 500) {
 throw new ApiError("Client error", response.status)
 }

 lastError = new ApiError("Server error", response.status)
 } catch (error) {
 lastError = error
 }

 if (attempt < maxRetries) {
 let delay = Math.pow(2, attempt) * 1000
```

```
 await new Promise(resolve => setTimeout(resolve, delay))
 }
 }

 throw lastError
}
```

Exponential backoff: wait 2s, 4s, 8s between retries.

Only retry on server errors and network errors. Never retry client errors.

GLOBAL ERROR HANDLER

```
class ApiClient {
 constructor(baseUrl) {
 this.baseUrl = baseUrl
 this.errorHandlers = []
 }

 onError(handler) {
 this.errorHandlers.push(handler)
 }

 async request(endpoint, options = {}) {
 let url = `${this.baseUrl}${endpoint}`

 try {
 let response = await fetch(url, options)

 if (!response.ok) {
 let data = await response.json().catch(() => ({}))
 throw new ApiError(data.message, response.status, data)
 }

 return response.json()
 } catch (error) {
```

```
 this.errorHandlers.forEach(handler => handler(error))
 throw error
 }
 }
}

let api = new ApiClient("https://api.example.com")

api.onError(error => {
 if (error.status === 401) {
 auth.logout()
 window.location.href = "/login"
 }
})

api.onError(error => {
 console.error("API Error:", error)
 trackError(error)
})
```

USER-FRIENDLY MESSAGES

```
function getErrorMessage(error) {
 if (error instanceof ApiError) {
 switch (error.status) {
 case 400:
 return "Invalid request. Please check your input."
 case 401:
 return "Please log in to continue."
 case 403:
 return "You do not have permission to do this."
 case 404:
 return "The requested item was not found."
 case 429:
 return "Too many requests. Please wait and try again."
 case 500:
 return "Server error. Please try again later."
```

```
 case 503:
 return "Service temporarily unavailable."
 default:
 return error.message ?? "Something went wrong."
 }
 }

 if (!navigator.onLine) {
 return "You appear to be offline."
 }

 return "Unable to connect. Check your internet connection."
}
```

FORM VALIDATION ERRORS

APIs often return validation errors for each field:

```
{
 "error": {
 "code": "VALIDATION_ERROR",
 "message": "Validation failed",
 "errors": [
 { "field": "email", "message": "Invalid email format" },
 { "field": "password", "message": "Password too short" }
 ]
 }
}
```

Handle them:

```
async function submitForm(formData) {
 try {
 await api.post("/users", formData)
 } catch (error) {
 if (error instanceof ApiError && error.data.errors) {
 let fieldErrors = {}
```

```
 error.data.errors.forEach(e => {
 fieldErrors[e.field] = e.message
 })

 return { success: false, errors: fieldErrors }
 }

 return { success: false, errors: { form: getErrorMessage(error) } }
 }

 return { success: true }
}
```

LOGGING AND MONITORING

Track errors for debugging:

```
async function apiRequest(url, options) {
 let startTime = Date.now()

 try {
 let response = await fetch(url, options)
 let duration = Date.now() - startTime

 logRequest({
 url,
 method: options.method ?? "GET",
 status: response.status,
 duration,
 success: response.ok
 })

 return response
 } catch (error) {
 let duration = Date.now() - startTime
```

```
 logRequest({
 url,
 method: options.method ?? "GET",
 error: error.message,
 duration,
 success: false
 })

 throw error
 }
}
```

GRACEFUL DEGRADATION

Show something even when API fails:

```
async function loadUserProfile() {
 let cachedProfile = localStorage.getItem("userProfile")

 try {
 let profile = await api.get("/users/me")

 localStorage.setItem("userProfile", JSON.stringify(profile))

 return profile
 } catch (error) {
 if (cachedProfile) {
 showMessage("Using cached data. Some info may be outdated.")
 return JSON.parse(cachedProfile)
 }

 throw error
 }
}
```

PRACTICE

Handle network errors and HTTP errors separately.

Create custom error classes with useful properties.

Implement retry logic with exponential backoff.

Show user-friendly error messages.

Log errors for monitoring.

WHAT YOU LEARNED

You learned to distinguish error types.
You learned HTTP error handling patterns.
You learned custom error classes.
You learned timeout and retry logic.
You learned user-friendly error messages.
You learned graceful degradation.

The next chapter teaches you about rate limiting and caching.
BOOK 12: WORKING WITH APIs
Chapter 7: Rate Limiting and Caching

UNDERSTANDING RATE LIMITS

APIs limit how many requests you can make.

Why:
- Prevent abuse
- Ensure fair usage

- Protect server resources
- Manage costs

Limits are usually:
- Per minute (60 requests/minute)
- Per hour (1000 requests/hour)
- Per day (10000 requests/day)

Exceeding limits returns 429 Too Many Requests.

RATE LIMIT HEADERS

APIs tell you your limit status in response headers:

X-RateLimit-Limit: 100
X-RateLimit-Remaining: 95
X-RateLimit-Reset: 1704067200

Limit is total allowed requests.
Remaining is requests left.
Reset is when the limit resets (Unix timestamp).

Read them:

```
let response = await fetch(url)

let limit = response.headers.get("X-RateLimit-Limit")
let remaining = response.headers.get("X-RateLimit-Remaining")
let resetTime = response.headers.get("X-RateLimit-Reset")

console.log(`${remaining}/${limit} requests remaining`)
```

HANDLING 429 ERRORS

```
async function fetchWithRateLimitHandling(url, options = {}) {
 let response = await fetch(url, options)

 if (response.status === = 429) {
 let retryAfter = response.headers.get("Retry-After")

 if (retryAfter) {
 let waitSeconds = parseInt(retryAfter, 10)
 console.log(`Rate limited. Waiting ${waitSeconds} seconds...`)
 await sleep(waitSeconds * 1000)
 return fetchWithRateLimitHandling(url, options)
 }

 throw new Error("Rate limit exceeded")
 }

 return response
}

function sleep(ms) {
 return new Promise(resolve = > setTimeout(resolve, ms))
}
```

REQUEST THROTTLING

Limit your own request rate:

```
class Throttler {
 constructor(maxPerSecond) {
 this.minInterval = 1000 / maxPerSecond
 this.lastRequest = 0
 }

 async throttle() {
 let now = Date.now()
 let timeSinceLastRequest = now - this.lastRequest
 let waitTime = this.minInterval - timeSinceLastRequest
```

```
 if (waitTime > 0) {
 await sleep(waitTime)
 }

 this.lastRequest = Date.now()
 }
}

let throttler = new Throttler(10)

async function throttledFetch(url) {
 await throttler.throttle()
 return fetch(url)
}
```

REQUEST QUEUE

Queue requests and process them at a controlled rate:

```
class RequestQueue {
 constructor(requestsPerSecond) {
 this.queue = []
 this.processing = false
 this.interval = 1000 / requestsPerSecond
 }

 add(requestFn) {
 return new Promise((resolve, reject) => {
 this.queue.push({ requestFn, resolve, reject })
 this.processQueue()
 })
 }

 async processQueue() {
 if (this.processing || this.queue.length === 0) {
 return
```

```
}

this.processing = true

while (this.queue.length > 0) {
let { requestFn, resolve, reject } = this.queue.shift()

try {
let result = await requestFn()
resolve(result)
} catch (error) {
reject(error)
}

if (this.queue.length > 0) {
await sleep(this.interval)
}
}

this.processing = false
}
}

let queue = new RequestQueue(5)

let result = await queue.add(() => fetch(url))
```

WHY CACHE API RESPONSES

Caching saves repeated requests for the same data.

Benefits:
- Faster response (no network delay)
- Reduced server load
- Works offline
- Saves bandwidth
- Avoids rate limits

IN-MEMORY CACHE

```
class SimpleCache {
 constructor(ttl = 60000) {
 this.cache = new Map()
 this.ttl = ttl
 }

 set(key, value) {
 this.cache.set(key, {
 value,
 expiry: Date.now() + this.ttl
 })
 }

 get(key) {
 let entry = this.cache.get(key)

 if (!entry) {
 return null
 }

 if (Date.now() > entry.expiry) {
 this.cache.delete(key)
 return null
 }

 return entry.value
 }

 has(key) {
 return this.get(key) !== null
 }

 clear() {
 this.cache.clear()
```

```
 }
}

let cache = new SimpleCache(5 * 60 * 1000)

async function fetchWithCache(url) {
 if (cache.has(url)) {
 return cache.get(url)
 }

 let response = await fetch(url)
 let data = await response.json()

 cache.set(url, data)

 return data
}
```

LOCALSTORAGE CACHE

```
class PersistentCache {
 constructor(prefix = "cache_", ttl = 3600000) {
 this.prefix = prefix
 this.ttl = ttl
 }

 getKey(key) {
 return this.prefix + key
 }

 set(key, value) {
 let entry = {
 value,
 expiry: Date.now() + this.ttl
 }

 localStorage.setItem(this.getKey(key), JSON.stringify(entry))
```

```
}

get(key) {
let stored = localStorage.getItem(this.getKey(key))

if (!stored) {
return null
}

let entry = JSON.parse(stored)

if (Date.now() > entry.expiry) {
localStorage.removeItem(this.getKey(key))
return null
}

return entry.value
}

clear() {
let keys = Object.keys(localStorage)

keys.forEach(key => {
if (key.startsWith(this.prefix)) {
localStorage.removeItem(key)
}
})
}
}
```

CACHE WITH STALE-WHILE-REVALIDATE

Return cached data immediately, fetch fresh data in background:

```
class SWRCache {
 constructor() {
 this.cache = new Map()
```

```
}

async get(key, fetcher, ttl = 60000) {
let entry = this.cache.get(key)
let now = Date.now()

if (entry) {
if (now > entry.staleTime) {
this.revalidate(key, fetcher, ttl)
}

return entry.value
}

let value = await fetcher()

this.cache.set(key, {
value,
staleTime: now + ttl
})

return value
}

async revalidate(key, fetcher, ttl) {
try {
let value = await fetcher()

this.cache.set(key, {
value,
staleTime: Date.now() + ttl
})
} catch (error) {
console.error("Revalidation failed:", error)
}
}
}

let swrCache = new SWRCache()
```

```
let users = await swrCache.get(
 "users",
 () => fetch("/api/users").then(r => r.json()),
 60000
)
```

HTTP CACHE HEADERS

Servers control caching with headers:

Cache-Control: max-age=3600
Cache the response for 1 hour.

Cache-Control: no-cache
Always validate with server before using cache.

Cache-Control: no-store
Never cache this response.

ETag: "abc123"
Unique identifier for this version of the resource.

Use ETag for conditional requests:

```
let response = await fetch(url, {
 headers: {
 "If-None-Match": previousEtag
 }
})

if (response.status === 304) {
 console.log("Resource unchanged, use cache")
}
```

CACHE INVALIDATION

When to clear cache:

After creating new data:

```
async function createUser(data) {
 let user = await api.post("/users", data)
 cache.delete("/users")
 return user
}
```

After updating data:

```
async function updateUser(id, data) {
 let user = await api.patch(`/users/${id}`, data)
 cache.delete(`/users/${id}`)
 cache.delete("/users")
 return user
}
```

After deleting data:

```
async function deleteUser(id) {
 await api.delete(`/users/${id}`)
 cache.delete(`/users/${id}`)
 cache.delete("/users")
}
```

CACHE STRATEGIES

Cache-first:
Check cache first. Only fetch if not cached.
Best for data that rarely changes.

Network-first:
Fetch first. Fall back to cache if network fails.

Best for data that must be fresh.

Stale-while-revalidate:
Return cached immediately. Refresh in background.
Best balance of speed and freshness.

Network-only:
Always fetch. Never cache.
For real-time data.

Cache-only:
Only use cache. Never fetch.
For offline mode.

COMPLETE CACHING CLIENT

```
class CachingApiClient {
 constructor(baseUrl) {
 this.baseUrl = baseUrl
 this.cache = new Map()
 this.defaultTtl = 5 * 60 * 1000
 }

 async get(endpoint, options = {}) {
 let cacheKey = endpoint
 let ttl = options.ttl ?? this.defaultTtl
 let skipCache = options.skipCache ?? false

 if (!skipCache) {
 let cached = this.getFromCache(cacheKey)
 if (cached !== null) {
 return cached
 }
 }

 let response = await fetch(`${this.baseUrl}${endpoint}`)
 let data = await response.json()
```

```
if (response.ok && ttl > 0) {
this.setCache(cacheKey, data, ttl)
}

return data
}

getFromCache(key) {
let entry = this.cache.get(key)

if (!entry) return null
if (Date.now() > entry.expiry) {
this.cache.delete(key)
return null
}

return entry.value
}

setCache(key, value, ttl) {
this.cache.set(key, {
value,
expiry: Date.now() + ttl
})
}

invalidate(pattern) {
if (typeof pattern === "string") {
this.cache.delete(pattern)
} else {
for (let key of this.cache.keys()) {
if (pattern.test(key)) {
this.cache.delete(key)
}
}
}
}

clearCache() {
```

```
 this.cache.clear()
 }
}
```

PRACTICE

Handle rate limit errors gracefully.

Implement request throttling.

Cache API responses appropriately.

Choose the right caching strategy.

Invalidate cache when data changes.

WHAT YOU LEARNED

You learned about rate limiting and how to handle 429 errors.
You learned request throttling and queuing.
You learned in-memory and persistent caching.
You learned stale-while-revalidate pattern.
You learned cache invalidation strategies.
You learned different caching strategies.

The next chapter teaches you RESTful patterns.
BOOK 12: WORKING WITH APIs
Chapter 8: RESTful Patterns

WHAT IS REST

REST stands for Representational State Transfer.

It is an architectural style for designing APIs.

RESTful APIs follow these principles:
- Resources identified by URLs
- Standard HTTP methods for actions
- Stateless communication
- Standard response formats

RESOURCES AND URLS

Everything is a resource. Resources have URLs.

Collection:
/users - All users
/products - All products
/orders - All orders

Single resource:
/users/123 - User with ID 123
/products/456 - Product with ID 456

Nested resources:
/users/123/orders - Orders for user 123
/orders/789/items - Items in order 789

Keep URLs simple and predictable.

CRUD OPERATIONS

CRUD maps to HTTP methods:

Create - POST /resources
Read - GET /resources or GET /resources/id
Update - PUT or PATCH /resources/id
Delete - DELETE /resources/id

Example:

POST /users - Create a user
GET /users - Get all users
GET /users/123 - Get user 123
PUT /users/123 - Replace user 123
PATCH /users/123 - Update user 123
DELETE /users/123 - Delete user 123

RESOURCE PATTERNS

List all resources:

GET /users

Response:

```
{
 "data": [
 { "id": 1, "name": "Alice" },
 { "id": 2, "name": "Bob" }
 ],
 "meta": {
 "total": 100,
 "page": 1,
 "perPage": 10
 }
}
```


Get single resource:

GET /users/1

Response:
{
 "id": 1,
 "name": "Alice",
 "email": "alice@example.com"
}

Create resource:

POST /users
Body: { "name": "Charlie", "email": "charlie@example.com" }

Response (201 Created):
{
 "id": 3,
 "name": "Charlie",
 "email": "charlie@example.com"
}

Update resource:

PATCH /users/3
Body: { "name": "Charles" }

Response:
{
 "id": 3,
 "name": "Charles",
 "email": "charlie@example.com"
}

Delete resource:

DELETE /users/3

Response: 204 No Content

FILTERING

Use query parameters to filter lists:

GET /users?status=active
GET /products?category=electronics&inStock=true
GET /orders?status=pending&minTotal=100

Multiple values:

GET /products?category=electronics,books
GET /users?id=1,2,3

SORTING

Use sort parameter:

GET /users?sort=name
GET /users?sort=-createdAt
GET /products?sort=price,-rating

Minus sign indicates descending order.

Multiple sort fields:

GET /users?sort=lastName,firstName

PAGINATION

Offset-based:

GET /users?page=2&limit=10
GET /users?offset=20&limit=10

Response includes pagination info:

```
{
 "data": [...],
 "pagination": {
 "page": 2,
 "limit": 10,
 "total": 100,
 "totalPages": 10,
 "hasNext": true,
 "hasPrev": true
 }
}
```

Cursor-based:

GET /users?cursor=abc123&limit=10

Response:

```
{
 "data": [...],
 "nextCursor": "xyz789",
 "prevCursor": "def456"
}
```

Cursor-based is better for large datasets.

FIELD SELECTION

Request only needed fields:

GET /users?fields=id,name,email
GET /products?fields=id,name,price

Reduces response size. Improves performance.

SEARCH

Use query parameter for search:

GET /users?q=alice
GET /products?search=laptop

More specific:

GET /users?name=alice&email=@example.com

RELATED RESOURCES

Include related data:

GET /users/123?include=orders,profile
GET /posts/456?include=author,comments

Response includes nested data:

```
{
 "id": 456,
 "title": "Hello World",
 "author": {
 "id": 1,
 "name": "Alice"
 },
 "comments": [
 { "id": 1, "text": "Great post!" }
```

```
 ]
}
```

ACTIONS ON RESOURCES

Sometimes you need actions that do not fit CRUD:

```
POST /users/123/activate
POST /orders/456/cancel
POST /emails/789/send
```

Use POST for actions. Put the action in the URL.

Or use a state change:

```
PATCH /users/123
Body: { "status": "active" }

PATCH /orders/456
Body: { "status": "cancelled" }
```

BATCH OPERATIONS

Create multiple resources:

```
POST /users/batch
Body: [
 { "name": "Alice" },
 { "name": "Bob" }
]
```

Delete multiple resources:

```
DELETE /users/batch
```

Body: { "ids": [1, 2, 3] }

Or with query:

DELETE /users?ids=1,2,3

ERROR RESPONSES

Standard error format:

```
{
 "error": {
 "code": "VALIDATION_ERROR",
 "message": "Validation failed",
 "details": [
 { "field": "email", "message": "Invalid format" }
 ]
 }
}
```

Or simpler:

```
{
 "error": "Not found",
 "message": "User with ID 123 does not exist"
}
```

Be consistent across your API.

VERSIONING

APIs evolve. Breaking changes need versioning.

URL versioning:

GET /v1/users
GET /v2/users

Header versioning:

GET /users
Accept: application/vnd.api+json; version=2

Query parameter:

GET /users?version=2

URL versioning is most common and clearest.

HATEOAS

Hypermedia As The Engine Of Application State.

Responses include links to related resources:

```
{
 "id": 123,
 "name": "Alice",
 "links": {
 "self": "/users/123",
 "orders": "/users/123/orders",
 "profile": "/users/123/profile"
 }
}
```

Clients follow links instead of constructing URLs.

Not all APIs implement HATEOAS.

REST CLIENT IMPLEMENTATION

```
class RestClient {
 constructor(baseUrl) {
 this.baseUrl = baseUrl
 }

 async request(method, endpoint, options = {}) {
 let url = new URL(endpoint, this.baseUrl)

 if (options.params) {
 Object.entries(options.params).forEach(([key, value]) => {
 url.searchParams.append(key, value)
 })
 }

 let config = {
 method,
 headers: {
 "Content-Type": "application/json",
 ...options.headers
 }
 }

 if (options.body) {
 config.body = JSON.stringify(options.body)
 }

 let response = await fetch(url, config)

 if (!response.ok) {
 throw await this.handleError(response)
 }

 if (response.status === 204) {
 return null
 }

 return response.json()
```

```
 }

 async handleError(response) {
 let data = await response.json().catch(() => ({}))
 let error = new Error(data.message ?? response.statusText)
 error.status = response.status
 error.data = data
 return error
 }

 list(resource, params = {}) {
 return this.request("GET", `/${resource}`, { params })
 }

 get(resource, id, params = {}) {
 return this.request("GET", `/${resource}/${id}`, { params })
 }

 create(resource, data) {
 return this.request("POST", `/${resource}`, { body: data })
 }

 update(resource, id, data) {
 return this.request("PATCH", `/${resource}/${id}`, { body: data })
 }

 replace(resource, id, data) {
 return this.request("PUT", `/${resource}/${id}`, { body: data })
 }

 delete(resource, id) {
 return this.request("DELETE", `/${resource}/${id}`)
 }
}

let api = new RestClient("https://api.example.com")

let users = await api.list("users", { page: 1, limit: 10 })
let user = await api.get("users", 123)
let newUser = await api.create("users", { name: "Alice" })
```

```
await api.update("users", 123, { name: "Updated" })
await api.delete("users", 123)
```

PRACTICE

Design URLs for resources.

Use proper HTTP methods for CRUD.

Implement filtering, sorting, and pagination.

Handle related resources.

Build a reusable REST client.

WHAT YOU LEARNED

You learned RESTful URL design.
You learned CRUD to HTTP method mapping.
You learned filtering and sorting patterns.
You learned pagination approaches.
You learned field selection and includes.
You learned batch operations.
You learned API versioning.

The next chapter teaches you about third-party APIs.

BOOK 12: WORKING WITH APIs

Chapter 9: Third-Party APIs

FINDING APIs

Many services provide public APIs:

Data APIs:
- OpenWeatherMap (weather)
- REST Countries (country data)
- NASA (space data)
- News API (news articles)
- CoinGecko (cryptocurrency)

Service APIs:
- Stripe (payments)
- Twilio (SMS/calls)
- SendGrid (email)
- Mapbox (maps)
- Firebase (backend services)

Social APIs:
- GitHub (repositories)
- Twitter/X (tweets)
- Discord (messaging)
- Spotify (music)

Find APIs at:
- RapidAPI.com
- PublicAPIs.dev
- GitHub awesome-apis lists

READING DOCUMENTATION

Every API has documentation. Read it.

Look for:
- Base URL
- Authentication method
- Available endpoints
- Request/response formats

- Rate limits
- Code examples

Example from OpenWeatherMap docs:

Base URL: https://api.openweathermap.org/data/2.5
Auth: API key as query parameter
Endpoint: /weather?q=city&appid=YOUR_KEY

OPENWEATHERMAP EXAMPLE

Get an API key at openweathermap.org (free tier available).

```
async function getWeather(city) {
 let apiKey = "your_api_key"
 let url = `https://api.openweathermap.org/data/2.5/weather?q=
${city}&appid=${apiKey}&units=metric`

 let response = await fetch(url)

 if (!response.ok) {
 throw new Error("City not found")
 }

 let data = await response.json()

 return {
 city: data.name,
 temperature: data.main.temp,
 description: data.weather[0].description,
 humidity: data.main.humidity
 }
}

let weather = await getWeather("Tokyo")

console.log(`${weather.city}: ${weather.temperature}C,
```

${weather.description}`)

REST COUNTRIES EXAMPLE

No API key needed.

```
async function getCountry(name) {
 let response = await fetch(`https://restcountries.com/v3.1/name/
${name}`)

 if (!response.ok) {
 throw new Error("Country not found")
 }

 let data = await response.json()
 let country = data[0]

 return {
 name: country.name.common,
 capital: country.capital?.[0] ?? "N/A",
 population: country.population,
 region: country.region,
 flag: country.flags.svg
 }
}

let japan = await getCountry("japan")

console.log(japan)
```

GITHUB API EXAMPLE

Public endpoints work without auth. Higher limits with token.

```
async function getUser(username) {
 let response = await fetch(`https://api.github.com/users/
${username}`)

 if (!response.ok) {
 throw new Error("User not found")
 }

 return response.json()
}

async function getRepos(username) {
 let response = await fetch(
 `https://api.github.com/users/${username}/repos?
sort=updated&per_page=10`
 )

 return response.json()
}

let user = await getUser("torvalds")
let repos = await getRepos("torvalds")

console.log(`${user.name} has ${user.public_repos} public repos`)

repos.forEach(repo => {
 console.log(`${repo.name}: ${repo.stargazers_count} stars`)
})
```

AUTHENTICATED GITHUB REQUESTS

For private repos or higher rate limits:

```
let token = "ghp_your_token"

async function githubApi(endpoint) {
 let response = await fetch(`https://api.github.com${endpoint}`, {
```

```
 headers: {
 "Authorization": `Bearer ${token}`,
 "Accept": "application/vnd.github.v3+json"
 }
 })

 return response.json()
}

let privateRepos = await githubApi("/user/repos?
visibility=private")
```

NEWS API EXAMPLE

Requires API key (free tier available at newsapi.org).

```
async function getTopHeadlines(country = "us") {
 let apiKey = "your_api_key"
 let url = `https://newsapi.org/v2/top-headlines?country=
${country}&apiKey=${apiKey}`

 let response = await fetch(url)
 let data = await response.json()

 return data.articles.map(article => ({
 title: article.title,
 source: article.source.name,
 url: article.url,
 publishedAt: new Date(article.publishedAt)
 }))
}

async function searchNews(query) {
 let apiKey = "your_api_key"
 let url = `https://newsapi.org/v2/everything?q=
${encodeURIComponent(query)}&apiKey=${apiKey}`
```

```
 let response = await fetch(url)
 let data = await response.json()

 return data.articles
}
```

COINGECKO API EXAMPLE

Cryptocurrency data. No API key needed.

```
async function getCryptoPrice(coinId) {
 let url = `https://api.coingecko.com/api/v3/simple/price?ids=
${coinId}&vs_currencies=usd`

 let response = await fetch(url)
 let data = await response.json()

 return data[coinId].usd
}

async function getCoinInfo(coinId) {
 let url = `https://api.coingecko.com/api/v3/coins/${coinId}`

 let response = await fetch(url)

 return response.json()
}

let btcPrice = await getCryptoPrice("bitcoin")

console.log(`Bitcoin: $${btcPrice}`)
```

HANDLING DIFFERENT RESPONSE FORMATS

Some APIs return data differently:

Wrapped in data object:

```
{
 "data": [...]
}
```

```
let items = response.data
```

Wrapped in results:

```
{
 "results": [...],
 "count": 100
}
```

```
let items = response.results
```

Direct array:

```
[...]
```

```
let items = response
```

With pagination:

```
{
 "items": [...],
 "next": "https://api.example.com/items?page=2"
}
```

BUILDING API WRAPPERS

Create reusable wrappers for APIs you use often:

```
class WeatherApi {
 constructor(apiKey) {
 this.apiKey = apiKey
 this.baseUrl = "https://api.openweathermap.org/data/2.5"
 }

 async getCurrentWeather(city, units = "metric") {
 let url = `${this.baseUrl}/weather?q=${city}&appid=
${this.apiKey}&units=${units}`
 let response = await fetch(url)

 if (!response.ok) {
 let error = await response.json()
 throw new Error(error.message ?? "Weather fetch failed")
 }

 return response.json()
 }

 async getForecast(city, units = "metric") {
 let url = `${this.baseUrl}/forecast?q=${city}&appid=
${this.apiKey}&units=${units}`
 let response = await fetch(url)

 if (!response.ok) {
 let error = await response.json()
 throw new Error(error.message ?? "Forecast fetch failed")
 }

 return response.json()
 }
}

let weather = new WeatherApi("your_api_key")

let current = await weather.getCurrentWeather("London")
let forecast = await weather.getForecast("London")
```

COMBINING MULTIPLE APIS

Build features using multiple APIs:

```
async function getTravelInfo(destination) {
 let [country, weather] = await Promise.all([
 getCountry(destination),
 getWeather(destination)
 ])

 return {
 destination: country.name,
 capital: country.capital,
 population: country.population,
 currentWeather: {
 temperature: weather.temperature,
 conditions: weather.description
 }
 }
}

let info = await getTravelInfo("Paris")

console.log(info)
```

API KEYS MANAGEMENT

For client-side apps, use a backend proxy:

Frontend:

```
async function getWeather(city) {
 let response = await fetch(`/api/weather?city=${city}`)
 return response.json()
```

```
}
```

Backend (Node.js example):

```
app.get("/api/weather", async (req, res) = > {
 let city = req.query.city
 let apiKey = process.env.WEATHER_API_KEY

 let response = await fetch(
 `https://api.openweathermap.org/data/2.5/weather?q=${city}
&appid=${apiKey}`
 )

 let data = await response.json()
 res.json(data)
})
```

The API key stays on the server, never exposed to browsers.

ERROR HANDLING FOR THIRD-PARTY APIS

```
async function safeApiCall(apiFunction, fallback = null) {
 try {
 return await apiFunction()
 } catch (error) {
 console.error("API call failed:", error.message)
 return fallback
 }
}

let weather = await safeApiCall(
 () = > getWeather("Tokyo"),
 { temperature: "N/A", description: "Unable to fetch weather" }
)
```

Always have fallbacks. External APIs can fail.

PRACTICE

Read API documentation carefully.

Start with simple public APIs.

Build wrappers for APIs you use often.

Handle errors and rate limits.

Keep API keys secure.

WHAT YOU LEARNED

You learned where to find public APIs.
You learned to read API documentation.
You learned to use weather, country, news, and cryptocurrency APIs.
You learned to build API wrappers.
You learned to combine multiple APIs.
You learned API key security.

The next chapter brings everything together with practice programs.
BOOK 12: WORKING WITH APIs
Chapter 10: Practice API Programs

COMPLETE API CLIENT

A full-featured reusable API client:

```
class ApiClient {
 #baseUrl
 #defaultHeaders
 #interceptors
 #cache
 #rateLimiter

 constructor(baseUrl, options = {}) {
 this.#baseUrl = baseUrl
 this.#defaultHeaders = {
 "Content-Type": "application/json",
 ...options.headers
 }
 this.#interceptors = {
 request: [],
 response: [],
 error: []
 }
 this.#cache = new Map()
 this.#rateLimiter = this.#createRateLimiter(options.rateLimit ??
10)
 }

 #createRateLimiter(maxPerSecond) {
 let lastRequest = 0
 let minInterval = 1000 / maxPerSecond

 return async () => {
 let now = Date.now()
 let waitTime = minInterval - (now - lastRequest)

 if (waitTime > 0) {
 await new Promise(r => setTimeout(r, waitTime))
 }

 lastRequest = Date.now()
 }
 }
```

```
addRequestInterceptor(fn) {
this.#interceptors.request.push(fn)
return this
}

addResponseInterceptor(fn) {
this.#interceptors.response.push(fn)
return this
}

addErrorInterceptor(fn) {
this.#interceptors.error.push(fn)
return this
}

async #applyInterceptors(type, value) {
let result = value

for (let interceptor of this.#interceptors[type]) {
result = await interceptor(result)
}

return result
}

async request(endpoint, options = {}) {
await this.#rateLimiter()

let cacheKey = options.method === "GET" ? endpoint : null

if (cacheKey && options.cache !== false) {
let cached = this.#cache.get(cacheKey)

if (cached && Date.now() < cached.expiry) {
return cached.data
}
}

let url = `${this.#baseUrl}${endpoint}`
```

```
let config = {
method: options.method ?? "GET",
headers: { ...this.#defaultHeaders, ...options.headers }
}

if (options.body) {
config.body = JSON.stringify(options.body)
}

if (options.params) {
let params = new URLSearchParams(options.params)
url = `${url}?${params}`
}

config = await this.#applyInterceptors("request", config)

let response
let retries = options.retries ?? 3

for (let attempt = 1; attempt <= retries; attempt++) {
try {
response = await fetch(url, config)
break
} catch (error) {
if (attempt === retries) {
throw await this.#applyInterceptors("error", error)
}
await new Promise(r => setTimeout(r, Math.pow(2, attempt) *
1000))
}
}

if (!response.ok) {
let errorData = await response.json().catch(() => ({}))
let error = new Error(errorData.message ?? response.statusText)
error.status = response.status
error.data = errorData
throw await this.#applyInterceptors("error", error)
}
```

```
let data = response.status === 204 ? null : await response.json()

data = await this.#applyInterceptors("response", data)

if (cacheKey && options.cacheTtl) {
this.#cache.set(cacheKey, {
data,
expiry: Date.now() + options.cacheTtl
})
}

return data
}

get(endpoint, options = {}) {
return this.request(endpoint, { ...options, method: "GET" })
}

post(endpoint, body, options = {}) {
return this.request(endpoint, { ...options, method: "POST", body })
}

put(endpoint, body, options = {}) {
return this.request(endpoint, { ...options, method: "PUT", body })
}

patch(endpoint, body, options = {}) {
return this.request(endpoint, { ...options, method: "PATCH", body
})
}

delete(endpoint, options = {}) {
return this.request(endpoint, { ...options, method: "DELETE" })
}

clearCache() {
this.#cache.clear()
}
}
```

WEATHER DASHBOARD

```
class WeatherDashboard {
 constructor(apiKey) {
 this.api = new ApiClient("https://api.openweathermap.org/
data/2.5")
 this.apiKey = apiKey
 }

 async getWeather(city) {
 return this.api.get("/weather", {
 params: {
 q: city,
 appid: this.apiKey,
 units: "metric"
 },
 cacheTtl: 10 * 60 * 1000
 })
 }

 async getForecast(city) {
 return this.api.get("/forecast", {
 params: {
 q: city,
 appid: this.apiKey,
 units: "metric"
 },
 cacheTtl: 30 * 60 * 1000
 })
 }

 formatWeather(data) {
 return {
 city: data.name,
 country: data.sys.country,
 temperature: Math.round(data.main.temp),
 feelsLike: Math.round(data.main.feels_like),
```

```
humidity: data.main.humidity,
description: data.weather[0].description,
icon: data.weather[0].icon,
wind: data.wind.speed
}
}

formatForecast(data) {
let daily = {}

data.list.forEach(item => {
let date = item.dt_txt.split(" ")[0]

if (!daily[date]) {
daily[date] = {
date,
temps: [],
descriptions: []
}
}

daily[date].temps.push(item.main.temp)
daily[date].descriptions.push(item.weather[0].description)
})

return Object.values(daily).map(day => ({
date: day.date,
high: Math.round(Math.max(...day.temps)),
low: Math.round(Math.min(...day.temps)),
description: day.descriptions[Math.floor(day.descriptions.length /
2)]
}))
}

async getDashboard(city) {
let [weather, forecast] = await Promise.all([
this.getWeather(city),
this.getForecast(city)
])
```

```
 return {
 current: this.formatWeather(weather),
 forecast: this.formatForecast(forecast)
 }
 }
}
```

DATA FETCHER WITH PAGINATION

```
class PaginatedFetcher {
 constructor(api, endpoint) {
 this.api = api
 this.endpoint = endpoint
 this.page = 1
 this.perPage = 20
 this.hasMore = true
 this.items = []
 }

 async fetchPage(page) {
 let data = await this.api.get(this.endpoint, {
 params: { page, per_page: this.perPage }
 })

 return {
 items: data.data ?? data.results ?? data,
 total: data.total ?? data.count ?? null,
 hasMore: data.hasMore ?? (data.data?.length === this.perPage)
 }
 }

 async loadMore() {
 if (!this.hasMore) {
 return []
 }

 let result = await this.fetchPage(this.page)
```

```
 this.items = [...this.items, ...result.items]
 this.hasMore = result.hasMore
 this.page++

 return result.items
 }

 async loadAll(maxPages = 10) {
 let allItems = []
 let page = 1

 while (page <= maxPages) {
 let result = await this.fetchPage(page)

 allItems = [...allItems, ...result.items]

 if (!result.hasMore) {
 break
 }

 page++
 }

 return allItems
 }

 reset() {
 this.page = 1
 this.hasMore = true
 this.items = []
 }
}
```

SEARCH WITH DEBOUNCE

```
class SearchService {
```

```
constructor(api, endpoint) {
this.api = api
this.endpoint = endpoint
this.cache = new Map()
this.pendingRequest = null
this.debounceTimeout = null
}

async search(query, options = {}) {
let delay = options.debounce ?? 300

if (this.debounceTimeout) {
clearTimeout(this.debounceTimeout)
}

if (this.pendingRequest) {
this.pendingRequest.abort()
}

if (this.cache.has(query)) {
return this.cache.get(query)
}

return new Promise((resolve, reject) => {
this.debounceTimeout = setTimeout(async () => {
let controller = new AbortController()
this.pendingRequest = controller

try {
let results = await this.api.get(this.endpoint, {
params: { q: query, ...options.params },
signal: controller.signal
})

this.cache.set(query, results)
resolve(results)
} catch (error) {
if (error.name !== "AbortError") {
reject(error)
}
```

```
 } finally {
 this.pendingRequest = null
 }
 }, delay)
 })
 }

 clearCache() {
 this.cache.clear()
 }
}
```

MULTI-API DATA AGGREGATOR

```
class DataAggregator {
 constructor() {
 this.sources = new Map()
 }

 addSource(name, fetchFn) {
 this.sources.set(name, fetchFn)
 return this
 }

 async fetchAll(options = {}) {
 let timeout = options.timeout ?? 10000
 let results = {}
 let errors = {}

 let promises = Array.from(this.sources.entries()).map(
 async ([name, fetchFn]) => {
 try {
 let controller = new AbortController()
 let timeoutId = setTimeout(() => controller.abort(), timeout)

 let data = await fetchFn(controller.signal)
```

```
 clearTimeout(timeoutId)

 results[name] = data
 } catch (error) {
 errors[name] = error.message
 }
 }
 )

 await Promise.all(promises)

 return { results, errors, hasErrors: Object.keys(errors).length > 0 }
 }

 async fetchWithFallback(primaryName, fallbackName) {
 let primaryFetch = this.sources.get(primaryName)
 let fallbackFetch = this.sources.get(fallbackName)

 try {
 return await primaryFetch()
 } catch {
 return await fallbackFetch()
 }
 }
}

let aggregator = new DataAggregator()

aggregator
 .addSource("weather", async () => {
 let response = await fetch("https://api.weather.com/current")
 return response.json()
 })
 .addSource("news", async () => {
 let response = await fetch("https://api.news.com/top")
 return response.json()
 })
 .addSource("crypto", async () => {
 let response = await fetch("https://api.crypto.com/prices")
 return response.json()
```

```
 })

let dashboard = await aggregator.fetchAll()
```

OFFLINE-FIRST DATA MANAGER

```
class OfflineDataManager {
 constructor(api, storageKey) {
 this.api = api
 this.storageKey = storageKey
 this.pendingSync = []
 }

 getLocalData() {
 let stored = localStorage.getItem(this.storageKey)
 return stored ? JSON.parse(stored) : null
 }

 setLocalData(data) {
 localStorage.setItem(this.storageKey, JSON.stringify(data))
 }

 getPendingSync() {
 let stored = localStorage.getItem(`${this.storageKey}_pending`)
 return stored ? JSON.parse(stored) : []
 }

 setPendingSync(items) {
 localStorage.setItem(`${this.storageKey}_pending`,
JSON.stringify(items))
 }

 async fetch(endpoint, options = {}) {
 try {
 let data = await this.api.get(endpoint, options)

 this.setLocalData({ data, timestamp: Date.now() })
```

```
return data
} catch {
let local = this.getLocalData()

if (local) {
return local.data
}

throw new Error("No data available offline")
}
}

async create(endpoint, item) {
if (!navigator.onLine) {
let pending = this.getPendingSync()
pending.push({ action: "create", endpoint, item, id: Date.now() })
this.setPendingSync(pending)

let local = this.getLocalData()

if (local) {
local.data.push({ ...item, _pending: true })
this.setLocalData(local)
}

return { ...item, _pending: true }
}

return this.api.post(endpoint, item)
}

async syncPending() {
if (!navigator.onLine) {
return { synced: 0, failed: 0 }
}

let pending = this.getPendingSync()
let synced = 0
let failed = 0
```

```
 let remaining = []

 for (let item of pending) {
 try {
 if (item.action === "create") {
 await this.api.post(item.endpoint, item.item)
 }

 synced++
 } catch {
 remaining.push(item)
 failed++
 }
 }

 this.setPendingSync(remaining)

 return { synced, failed }
 }
}
```

WHAT YOU BUILT

You built five complete programs:

Full-featured API client with caching and rate limiting
Weather dashboard with current and forecast data
Paginated data fetcher with load more
Search service with debounce and caching
Offline-first data manager with sync

Each program demonstrated:

Classes with private fields
Async/await patterns
Error handling
Caching strategies

Rate limiting
Offline support

WHAT COMES NEXT

You now understand working with APIs:

- What APIs are and how they work (Book 12, Chapter 1)
- HTTP basics (Book 12, Chapter 2)
- The Fetch API (Book 12, Chapter 3)
- Working with JSON (Book 12, Chapter 4)
- Authentication (Book 12, Chapter 5)
- Error handling (Book 12, Chapter 6)
- Rate limiting and caching (Book 12, Chapter 7)
- RESTful patterns (Book 12, Chapter 8)
- Third-party APIs (Book 12, Chapter 9)

Book 13 teaches you Node.js. You will:

- Run JavaScript on servers
- Build command-line tools
- Work with files and processes
- Create HTTP servers
- Build backend APIs

Node.js takes JavaScript beyond the browser.

You finished Book 12. You connect to the world. Now you build the world.